001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.bag;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.util.Collection;  
023import java.util.Iterator;  
024  
025import org.apache.commons.collections4.Bag;  
026  
027/\*\*  
028 \* Decorates another {@link Bag} to comply with the Collection contract.  
029 \* <p>  
030 \* By decorating an existing {@link Bag} instance with a {@link CollectionBag},  
031 \* it can be safely passed on to methods that require Collection types that  
032 \* are fully compliant with the Collection contract.  
033 \* </p>  
034 \* <p>  
035 \* The method javadoc highlights the differences compared to the original Bag interface.  
036 \* </p>  
037 \*  
038 \* @see Bag  
039 \* @param <E> the type of elements in this bag  
040 \* @since 4.0  
041 \*/  
042public final class CollectionBag<E> extends AbstractBagDecorator<E> {  
043  
044 /\*\* Serialization version \*/  
045 private static final long serialVersionUID = -2560033712679053143L;  
046  
047 /\*\*  
048 \* Factory method to create a bag that complies to the Collection contract.  
049 \*  
050 \* @param <E> the type of the elements in the bag  
051 \* @param bag the bag to decorate, must not be null  
052 \* @return a Bag that complies to the Collection contract  
053 \* @throws NullPointerException if bag is null  
054 \*/  
055 public static <E> Bag<E> collectionBag(final Bag<E> bag) {  
056 return new CollectionBag<>(bag);  
057 }  
058  
059 //-----------------------------------------------------------------------  
060 /\*\*  
061 \* Constructor that wraps (not copies).  
062 \*  
063 \* @param bag the bag to decorate, must not be null  
064 \* @throws NullPointerException if bag is null  
065 \*/  
066 public CollectionBag(final Bag<E> bag) {  
067 super(bag);  
068 }  
069  
070 //-----------------------------------------------------------------------  
071 /\*\*  
072 \* Write the collection out using a custom routine.  
073 \*  
074 \* @param out the output stream  
075 \* @throws IOException if an error occurs while writing to the stream  
076 \*/  
077 private void writeObject(final ObjectOutputStream out) throws IOException {  
078 out.defaultWriteObject();  
079 out.writeObject(decorated());  
080 }  
081  
082 /\*\*  
083 \* Read the collection in using a custom routine.  
084 \*  
085 \* @param in the input stream  
086 \* @throws IOException if an error occurs while reading from the stream  
087 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
088 \* @throws ClassCastException if deserialised object has wrong type  
089 \*/  
090 @SuppressWarnings("unchecked") // will throw CCE, see Javadoc  
091 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
092 in.defaultReadObject();  
093 setCollection((Collection<E>) in.readObject());  
094 }  
095  
096 //-----------------------------------------------------------------------  
097 // Collection interface  
098 //-----------------------------------------------------------------------  
099  
100 /\*\*  
101 \* <i>(Change)</i>  
102 \* Returns <code>true</code> if the bag contains all elements in  
103 \* the given collection, <b>not</b> respecting cardinality. That is,  
104 \* if the given collection <code>coll</code> contains at least one of  
105 \* every object contained in this object.  
106 \*  
107 \* @param coll the collection to check against  
108 \* @return <code>true</code> if the Bag contains at least one of every object in the collection  
109 \*/  
110 @Override  
111 public boolean containsAll(final Collection<?> coll) {  
112 final Iterator<?> e = coll.iterator();  
113 while (e.hasNext()) {  
114 if(!contains(e.next())) {  
115 return false;  
116 }  
117 }  
118 return true;  
119 }  
120  
121 /\*\*  
122 \* <i>(Change)</i>  
123 \* Adds one copy of the specified object to the Bag.  
124 \* <p>  
125 \* Since this method always increases the size of the bag, it  
126 \* will always return <code>true</code>.  
127 \*  
128 \* @param object the object to add  
129 \* @return <code>true</code>, always  
130 \*/  
131 @Override  
132 public boolean add(final E object) {  
133 return add(object, 1);  
134 }  
135  
136 @Override  
137 public boolean addAll(final Collection<? extends E> coll) {  
138 boolean changed = false;  
139 final Iterator<? extends E> i = coll.iterator();  
140 while (i.hasNext()) {  
141 final boolean added = add(i.next(), 1);  
142 changed = changed || added;  
143 }  
144 return changed;  
145 }  
146  
147 /\*\*  
148 \* <i>(Change)</i>  
149 \* Removes the first occurrence of the given object from the bag.  
150 \* <p>  
151 \* This will also remove the object from the {@link #uniqueSet()} if the  
152 \* bag contains no occurrence anymore of the object after this operation.  
153 \*  
154 \* @param object the object to remove  
155 \* @return <code>true</code> if this call changed the collection  
156 \*/  
157 @Override  
158 public boolean remove(final Object object) {  
159 return remove(object, 1);  
160 }  
161  
162 /\*\*  
163 \* <i>(Change)</i>  
164 \* Remove all elements represented in the given collection,  
165 \* <b>not</b> respecting cardinality. That is, remove <i>all</i>  
166 \* occurrences of every object contained in the given collection.  
167 \*  
168 \* @param coll the collection to remove  
169 \* @return <code>true</code> if this call changed the collection  
170 \*/  
171 @Override  
172 public boolean removeAll(final Collection<?> coll) {  
173 if (coll != null) {  
174 boolean result = false;  
175 final Iterator<?> i = coll.iterator();  
176 while (i.hasNext()) {  
177 final Object obj = i.next();  
178 final boolean changed = remove(obj, getCount(obj));  
179 result = result || changed;  
180 }  
181 return result;  
182 }  
183 // let the decorated bag handle the case of null argument  
184 return decorated().removeAll(null);  
185 }  
186  
187 /\*\*  
188 \* <i>(Change)</i>  
189 \* Remove any members of the bag that are not in the given collection,  
190 \* <i>not</i> respecting cardinality. That is, any object in the given  
191 \* collection <code>coll</code> will be retained in the bag with the same  
192 \* number of copies prior to this operation. All other objects will be  
193 \* completely removed from this bag.  
194 \* <p>  
195 \* This implementation iterates over the elements of this bag, checking  
196 \* each element in turn to see if it's contained in <code>coll</code>.  
197 \* If it's not contained, it's removed from this bag. As a consequence,  
198 \* it is advised to use a collection type for <code>coll</code> that provides  
199 \* a fast (e.g. O(1)) implementation of {@link Collection#contains(Object)}.  
200 \*  
201 \* @param coll the collection to retain  
202 \* @return <code>true</code> if this call changed the collection  
203 \*/  
204 @Override  
205 public boolean retainAll(final Collection<?> coll) {  
206 if (coll != null) {  
207 boolean modified = false;  
208 final Iterator<E> e = iterator();  
209 while (e.hasNext()) {  
210 if (!coll.contains(e.next())) {  
211 e.remove();  
212 modified = true;  
213 }  
214 }  
215 return modified;  
216 }  
217 // let the decorated bag handle the case of null argument  
218 return decorated().retainAll(null);  
219 }  
220  
221 //-----------------------------------------------------------------------  
222 // Bag interface  
223 //-----------------------------------------------------------------------  
224  
225 /\*\*  
226 \* <i>(Change)</i>  
227 \* Adds <code>count</code> copies of the specified object to the Bag.  
228 \* <p>  
229 \* Since this method always increases the size of the bag, it  
230 \* will always return <code>true</code>.  
231 \*  
232 \* @param object the object to add  
233 \* @param count the number of copies to add  
234 \* @return <code>true</code>, always  
235 \*/  
236 @Override  
237 public boolean add(final E object, final int count) {  
238 decorated().add(object, count);  
239 return true;  
240 }  
241  
242}